

Ordering Metro Lines by Block Crossings

Martin Fink¹ and Sergey Pupyrev²

¹ Lehrstuhl für Informatik I, Universität Würzburg, Germany.

² University of Arizona, USA.

Abstract. A problem that arises in drawings of transportation networks is to minimize the number of crossings between different transportation lines. While this can be done efficiently under specific constraints, not all solutions are visually equivalent. We suggest to merge crossings into *block crossings*, that is, crossings of two neighboring groups of consecutive lines. Unfortunately, minimizing the total number of block crossings is NP-hard even for very simple graphs. We give approximation algorithms for special classes of graphs and an asymptotically worst-case optimal algorithm for block crossings on general graphs. That is, we bound the number of block crossings that our algorithm needs and construct worst-case instances on which the number of block crossings that is necessary in any solution is asymptotically the same as our bound.

1 Introduction

In many metro maps and transportation networks some edges, that is, railway track or road segments, are used by several *lines*. Usually, to visualize such networks, lines that share an edge are drawn individually along the edge in distinct colors. Often, some lines must cross, and it is desirable to draw the lines with few crossings. The *metro-line crossing minimization* problem has recently been introduced [4]. The goal is to order the lines along each edge such that the number of crossings is minimized. So far, the focus has been on the number of crossings and not on their visualization, although two line orders with the same crossing number may look quite differently; see Fig. 1.

Our aim is to improve the readability of metro maps by computing line orders that are aesthetically more pleasing. To this end, we merge *pairwise* crossings into crossings of blocks of lines minimizing the number of *block crossings* in the map. Informally, a block crossing is an intersection of two neighboring groups of consecutive lines sharing the same edge; see Fig. 1(b). We consider two variants of the problem. In the first variant, we want to find a line ordering with the minimum number of block crossings. In the second variant, we want to minimize both pairwise and block crossings.

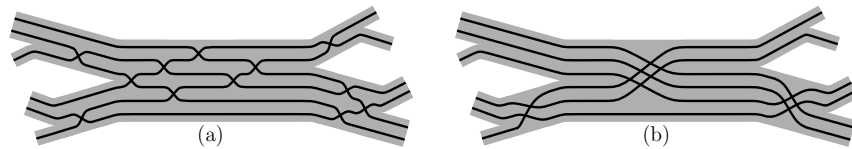


Fig. 1. Optimal orderings of a metro network: (a) 12 pairwise crossings; (b) 3 block crossings.

Motivation. Although we present our results in terms of the classic problem of visualizing metro maps, crossing minimization between paths on an embedded graph is used in various fields. In very-large-scale integration (VLSI) chip layout, a wire diagram should have few wire crossings [9]. Another application is the visualization of biochemical pathways [14]. In graph drawing the number of edge crossings is considered one of the most popular aesthetic criteria. Recently, a lot of research, both in graph drawing and information visualization, is devoted to *edge bundling*. In this setting, some edges are drawn close together—like metro lines—which emphasizes the structure of the graph [13]. Block crossings can greatly improve the readability of bundled graph drawings.

Problem definition. The input consists of an embedded graph $G = (V, E)$, and a set $L = \{l_1, \dots, l_{|L|}\}$ of simple paths in G . We call G the *underlying network* and the paths *lines*. The nodes of G are *stations* and the endpoints v_0, v_k of a line $(v_0, \dots, v_k) \in L$ are *terminals*. For each edge $e = (u, v) \in E$, let L_e be the set of lines passing through e . For $i \leq j < k$, a *block move* (i, j, k) on the sequence $\pi = [\pi_1, \dots, \pi_n]$ of lines is the exchange of two consecutive blocks π_i, \dots, π_j and π_{j+1}, \dots, π_k . We are interested in *line orders* $\pi^0(e), \dots, \pi^{t(e)}(e)$ on e , so that $\pi^0(e)$ is the order of lines L_e on e close to u , $\pi^{t(e)}(e)$ is the order close to v , and each $\pi^i(e)$ is an ordering of L_e so that $\pi^{i+1}(e)$ is constructed from $\pi^i(e)$ by a block move. We say that there are t *block crossings* on e .

Following previous work [1,12] we use the *edge crossings* model, that is, we do not hide crossings under station symbols if possible. Two lines sharing at least one common edge either do not cross or cross each other on an edge but never in a node; see Fig. 2(a). For pairs of lines sharing a vertex but no edges, crossings at the vertex are allowed and not counted as they exist at this position in any solution. We call them *unavoidable vertex crossings*; see Fig. 2(b). If the line orders on the edges incident to a vertex v produce only edge crossings and unavoidable vertex crossings, we call them *consistent* in v . Line orders for all edges are consistent if they are consistent in all nodes. More formally, we can check consistency of line orders in a vertex v by looking at each incident edge e . Close to v the order of lines L_e on e is fixed. The other edges e_1, \dots, e_k incident to v contain lines of L_e . The combined order of L_e on the edges e_1, \dots, e_k must be the same as the order on e ; otherwise, lines of L_e would cross in v . Now, we can define the *block crossing minimization* problem (BCM).

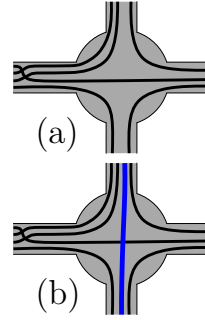


Fig. 2. Consistent line orders (a) without, (b) with an unavoidable vertex crossing.

Problem 1 (BCM). Let $G = (V, E)$ be an embedded graph and let L be a set of lines on G . For each edge $e \in E$, find line orders $\pi^0(e), \dots, \pi^{t(e)}(e)$ such that the total number of block crossings, $\sum_{e \in E} t(e)$, is minimum and the line orders are consistent.

In this paper, we restrict our attention to instances with two additional properties. First, any line terminates at nodes of degree one and no two lines terminate at the same

graph class	BCM		MBCM	
single edge	11/8-approximation	[7]	3-approximation	Section 2
path	3-approximation	Section 3	3-approximation	Section 3
tree	$\leq 2 L - 3$ crossings	Section 4	$\leq 2 L - 3$ crossings	Section 4
upward tree	—		6-approximation	Section 4
general graph	$O(L \sqrt{ E })$ crossings	Section 5	$O(L \sqrt{ E })$ crossings	Section 5

Table 1. Overview of our results for BCM and MBCM.

node (**path terminal property**). Second, the intersection of two lines, that is, the edges and vertices they have in common, forms a path (**path intersection property**). This includes the cases that the intersection is empty or a single node. If both properties hold, a pair of lines either has to cross, that is, a crossing is *necessary*, or it can be kept crossing-free, that is, a crossing is *unnecessary*. The orderings that are optimal with respect to pairwise crossings are exactly the orderings that contain just necessary crossings (Lemma 2 in [12]); that is, any pair of lines crosses at most once, in an equivalent formulation. As this is a very reasonable condition also for block crossings, we use it to define the *monotone block crossing minimization* problem (MBCM) whose feasible solutions must have the minimum number of pairwise crossings.

Problem 2 (MBCM). Given an instance of BCM find a feasible solution that minimizes the number of block crossings subject to the constraint that no two lines cross twice.

On some instances BCM does allow less crossings than MBCM does, see Fig. 3.

Our contribution. We introduce the new problems BCM and MBCM. To the best of our knowledge, ordering lines by block crossings is a new direction in graph drawing. So far BCM has been investigated only for the case that the *skeleton*, that is, the graph without terminals, is a single edge [2] while MBCM is a completely new problem.

We first analyze MBCM on a single edge (Section 2), exploiting, to some extent, the similarities to *sorting by transpositions* [2]. Then, we use the notion of *good pairs* of lines, that is, lines that should be neighbors, for developing an approximation algorithm for BCM on graphs whose skeleton is a path (Section 3); we properly define good pairs so that changes between adjacent edges are taken into account. Yet, good pairs can not always be kept close; we introduce a good strategy for breaking pairs when needed.

Unfortunately, the approximation algorithm does not generalize to trees. We do, however, develop a worst-case optimal algorithm for trees (Section 4). It needs $2|L| - 3$ block crossings and there are instances in which this number of block crossings is necessary in any solution. We then use our algorithm for obtaining approximate solutions for MBCM on the special class of *upward trees*.

As our main result, we develop an algorithm for obtaining a solution for (M)BCM on general graphs (Section 5). We show that it uses only monotone block moves and analyze the upper bound on the number of block crossings. While the algorithm itself is simple and easy to implement, proving the upper bound is non-trivial. Next, we show that the bound is tight; we use a result from projective geometry for constructing worst-

case examples in which any feasible solution contains many block crossings. Hence, our algorithm is asymptotically worst-case optimal. Table 1 summarizes our results.

Related work. Line crossing problems in transportation networks were initiated by Benkert et al. [4], who considered the problem of *metro-line crossing minimization* (MLCM) on a single edge. MLCM in its general model is challenging; its complexity is open and no efficient algorithms are known for the case of two or more edges. Bekos et al. [3] addressed the problem on paths and trees. They also proved that a variant in which all lines must be placed outermost in their terminals is NP-hard. Subsequently, Argyriou et al. [1] and Nöllenburg [12] devised polynomial-time algorithms for general graphs with the path terminal property. Pupyrev et al. [13] studied MLCM in the context of edge bundling. They suggested a linear-time algorithm for MLCM on instances with the path terminal property. All these works are dedicated to pairwise crossings; the optimization criterion being the number of crossing pairs of lines.

A closely related problem arises in VLSI design, where the goal is to minimize intersections between nets (physical wires) [9,11]. Net patterns with less crossings most likely have better electrical characteristics and require less wiring area; hence, it is an important optimization criterion in circuit board design. Marek-Sadowska and Sarrafzadeh [11] considered not only minimizing the number of crossings, but also suggested to distribute the crossings among the circuit regions in order to simplify net routing.

BCM on a *single* edge is equivalent to the problem of sorting a permutation by block moves, which is well studied in computational biology for DNA sequences; it is known as *sorting by transpositions* [2,6]. The task is to find the shortest sequence of block moves transforming a given permutation into the identity permutation. The complexity of the problem was open for a long time; only very recently it has been shown to be NP-hard [5]. The currently best known algorithm has an approximation ratio of $11/8$ [7]. The proof of correctness of that algorithm is based on a computer analysis which verifies more than 80,000 configurations. To the best of our knowledge, no tight upper bound is known for the problem. There are several variants of sorting by transpositions; see the survey of Fertin et al. [8]. For instance, Vergara et al. [10] used *correcting short block moves* to sort a permutation. In our terminology, these are monotone moves such that the combined length of exchanged blocks does not exceed three. Hence, their problem is a restricted variant of MBCM on a single edge. We notice that the complexity of this variant is still unknown.

2 Block Crossings on a Single Edge

First, we restrict our attention to networks consisting of a single edge with multiple lines passing through it. BCM then can be reformulated as follows. Given two permutations π and τ (determined by the order of terminals on both sides of the edge), find the shortest sequence of block moves transforming π into τ . By relabeling we can assume that τ is the identity permutation, and the goal is to sort π . This problem is known as *sorting by transpositions* [2]. We concentrate on the new problem of sorting with monotone block moves; that means that the relative order of any pair of elements changes at most once. The problems are not equivalent; see Fig.3 for an example where non-monotonicity

allows for fewer crossings. In what follows, we give lower and upper bounds on the number of block crossings for MBCM on a single edge. Additionally, we present a simple 3-approximation algorithm for the problem.

We first introduce some terminology following the one from previous works where possible. Let $\pi = [\pi_1, \dots, \pi_n]$ be a permutation of n elements. For convenience, we assume there are extra elements $\pi_0 = 0$ and $\pi_{n+1} = n + 1$ at the beginning of the permutation and at the end, respectively. A *block* in π is a sequence of consecutive elements π_i, \dots, π_j with $i \leq j$. A *block move* (i, j, k) with $i \leq j < k$ on π maps $[\dots \pi_{i-1} \pi_i \dots \pi_j \pi_{j+1} \dots \pi_k \pi_{k+1} \dots]$ to $[\dots \pi_{i-1} \pi_{j+1} \dots \pi_k \pi_i \dots \pi_j \pi_{k+1} \dots]$. We say that a block move (i, j, k) is *monotone* if $\pi_q > \pi_r$ for all $i \leq q \leq j < r \leq k$. We denote the minimum number of monotone block moves needed to sort π by $\text{bc}(\pi)$. An ordered pair (π_i, π_{i+1}) is a *good pair* if $\pi_{i+1} = \pi_i + 1$, and a *breakpoint* otherwise. Intuitively, sorting π is a process of creating good pairs (or destroying breakpoints) by block moves. We call a permutation *simple* if it has no good pairs. Any permutation can be uniquely simplified—by gluing good pairs together and relabeling—without affecting its distance to the identity permutation [6]. A breakpoint (π_i, π_{i+1}) is a *descent* if $\pi_i > \pi_{i+1}$, and a *gap* otherwise. We use $\text{bp}(\pi)$, $\text{des}(\pi)$, and $\text{gap}(\pi)$ to denote the number of breakpoints, descents, and gaps in π . The *inverse* of a permutation π is the permutation π^{-1} in which each element and the index of its position are exchanged, that is, $\pi_{\pi_i}^{-1} = i$ for $1 \leq i \leq n$. A descent in π^{-1} , that is, a pair of elements $\pi_i = \pi_j + 1$ with $i < j$, is called an *inverse descent* in π . Analogously, an *inverse gap* is a pair of elements $\pi_i = \pi_j + 1$ with $i > j + 1$. Now, we give lower and upper bounds for MBCM, that is, on $\text{bc}(\pi)$.

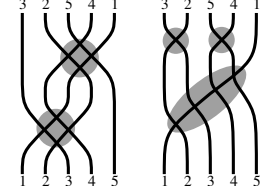


Fig. 3. Permutation $[3\ 2\ 5\ 4\ 1]$ is sorted with 2 block moves and 3 monotone block moves.

A lower bound. It is easy to see that a block move affects three pairs of adjacent elements. Therefore the number of breakpoints can be reduced by at most three in a move. As only the identity permutation has no breakpoints, this implies $\text{bc}(\pi) \geq \text{bp}(\pi)/3$ for a simple permutation π . The following observations yield better lower bounds.

Lemma 1. *In a monotone block move, the number of descents in a permutation decreases by at most one, and the number of gaps decreases by at most two.*

Proof. Consider a monotone move $[\dots ab \dots cd \dots ef \dots] \Rightarrow [\dots ad \dots eb \dots cf \dots]$; it affects three adjacencies. Suppose a descent is destroyed between a and b , that is, $a > b$ and $a < d$. Then, $b < d$ which contradicts monotonicity. Similarly, no descent can be destroyed between e and f . On the other hand, since $c > d$, no gap can be destroyed between c and d . \square

A similar claim holds for the inverse descents and gaps.

Lemma 2. *In a monotone block move, the number of inverse descents decreases by at most one, and the number of inverse gaps decreases by at most two.*

Proof. Consider a monotone exchange of blocks π_i, \dots, π_j and π_{j+1}, \dots, π_k . Note that inverse descents can only be destroyed between elements π_q ($i \leq q \leq j$) and π_r ($j+1 \leq r \leq k$). Suppose that the move destroys two inverse descents such that the first block contains elements $x+1$ and $y+1$, and the second block contains x and y . Since the block move is monotone, $y+1 > x$ and $x+1 > y$, which means that $x = y$.

On the other hand, there cannot be inverse gaps between elements π_q ($i \leq q \leq j$) and π_r ($j+1 \leq r \leq k$). Therefore, there are only two possible inverse gaps between π_{i-1} and π_r ($j < r \leq k$), and between π_q ($i \leq q \leq j$) and π_{k+1} . \square

Combining the lemmas, we obtain the following result.

Theorem 1. *A lower bound on the number of monotone block moves needed to sort a permutation is $\text{bc}(\pi) \geq \max(\text{bp}(\pi)/3, \text{des}(\pi), \text{gap}(\pi)/2, \text{des}(\pi^{-1}), \text{gap}(\pi^{-1})/2)$.*

An upper bound. We suggest the following algorithm for sorting a simple permutation π : In each step find the smallest i such that $\pi_i \neq i$ and move element i to position i , that is, exchange block π_i, \dots, π_{k-1} and π_k , where $\pi_k = i$. Clearly, the step destroys at least one breakpoint. Therefore $\text{bc}(\pi) \leq \text{bp}(\pi)$ and the algorithm yields a 3-approximation.

Theorem 2. *We can find a 3-approximation for MBCM on a single edge in $O(n^2)$ time.*

To construct a better upper bound, we first consider a constrained sorting problem in which at least one of the moved blocks has unit size; that is, we allow only block moves of types (i, i, k) and $(i, k-1, k)$. Let $\text{bc}^1(\pi)$ be the minimum number of such block moves needed to sort π . We show how to compute $\text{bc}^1(\pi)$ exactly. An *increasing subsequence* of π is a sequence $\pi_{l_1}, \pi_{l_2}, \dots$ such that $\pi_{l_1} < \pi_{l_2} < \dots$ and $l_1 < l_2 < \dots$. Let $\text{lis}(\pi)$ be the size of the longest increasing subsequence of π .

Lemma 3. $\text{bc}^1(\pi) = n - \text{lis}(\pi)$.

Proof. $\text{bc}^1(\pi) \geq n - \text{lis}(\pi)$. Consider a monotone move $\sigma = (i, i, k)$ in π ($\sigma = (i, k-1, k)$ is symmetric). Let $\tilde{\pi} = [\pi_1, \dots, \pi_{i-1}, \pi_{i+1}, \dots, \pi_n]$ be the permutation π without element π_i . Clearly, $\text{lis}(\tilde{\pi}) \leq \text{lis}(\pi)$. If we apply σ , the resulting permutation $\sigma\pi = [\pi_1, \dots, \pi_{i-1}, \pi_{i+1}, \dots, \pi_k, \pi_i, \pi_{k+1}, \dots, \pi_n]$ has one extra element compared to $\tilde{\pi}$, and, therefore, $\text{lis}(\sigma\pi) \leq \text{lis}(\tilde{\pi}) + 1$. Hence, $\text{lis}(\sigma\pi) \leq \text{lis}(\pi) + 1$, that is, the length of the longest increasing subsequence cannot increase by more than one in a move. The inequality follows since $\text{lis}(\tau) = n$ for the identity permutation τ .

$\text{bc}^1(\pi) \leq n - \text{lis}(\pi)$. Let $S = [\dots s_1 \dots s_2 \dots s_{\text{lis}} \dots]$ be a fixed longest increasing subsequence in π . We show how to choose a move that increases the length of S . Let $\pi_i \notin S$ be the rightmost element (that is, i is maximum) lying between elements s_j and s_{j+1} of S so that $\pi_i > s_{j+1}$. We move π_i rightwards to its proper position p_i inside S . This is a monotone move, as π_i was chosen rightmost. If no such element p_i exists, we symmetrically choose the leftmost p_i with $p_i < s_j$ and bring it into its proper position in S . In both cases S grows. \square

Corollary 1. *Any permutation can be sorted by $n - \text{lis}(\pi)$ monotone block moves.*

3 Block Crossings on a Path

Now we consider an embedded graph $G = (V, E)$ consisting of a path $P = (V_P, E_P)$ with attached terminals. In every node $v \in V_P$ the clockwise order of terminals adjacent to v is given, and we assume the path is oriented from left to right. We say that a line l starts at v if v is the leftmost vertex on P that lies on l and ends at its rightmost vertex of the path. As we consider only crossings of lines sharing an edge, we assume that the terminals connected to any path node v are in such an order that first lines end at v and then lines start at v , as shown in Fig. 4(b).

We suggest a 3-approximation algorithm for BCM. Similar to the single edge case, the basic idea of the algorithms is to consider good pairs of lines. A *good pair* is an ordered pair of lines that will be adjacent—in this order—in any feasible solution when one of the lines ends. We argue that our algorithm creates at least one additional good pair per block crossing, while even the optimum creates at most three new good pairs per crossing. To describe our algorithm we first define a *good pair*.

Definition 1 (Good pair).

- (i) If two lines a and b end on the same node, and a and b are consecutive in clockwise order, then (a, b) is a good pair (as it is in the case of a single edge in Section 2).
- (ii) Let v be a node with edges (u, v) and (v, w) on P , let a_1 be the first line starting on v above P , and let a_2 be the last line ending on v above P as in Fig. 4(a). If (a_1, b) is a good pair, then (a_2, b) also is a good pair. We say that (a_2, b) is inherited from (a_1, b) , and identify (a_1, b) with (a_2, b) , which is possible as a_1 and a_2 do not share an edge. Analogously, there is inheritance for lines starting/ending below P .

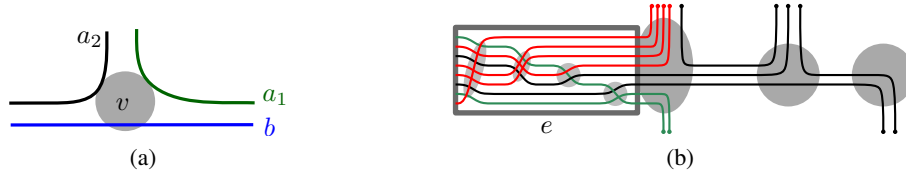


Fig. 4. (a) Inheritance of a good pair above node v . (b) Ordering the lines on edge e in a step of the algorithm.

As a preprocessing step, we add a virtual line $t_e(b_e)$ for each edge $e \in E_P$. The line $t_e(b_e)$ is the last line starting before e , and the first line ending after e to the top (bottom). Although virtual lines are never moved, $t_e(b_e)$ does participate in good pairs, which model the fact that the first lines ending after an edge should be brought to the top (bottom).

There are important properties of good pairs.

Lemma 4. *On an edge $e \in E_P$ there is, for each line l , at most one good pair (l', l) and at most one good pair (l, l'') .*

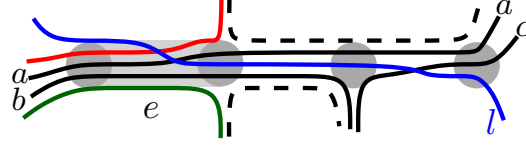


Fig. 5. The (necessary) insertion of line l forces breaking the good pair $(a, b) (\equiv (a, c))$ on edge e .

Proof. Let e be the rightmost edge where there is a line l that violates the property, that is, there are two good pairs (l', l) and (l'', l) (symmetrically for l on the top). If l ends after e there clearly can be at most one of these good pairs. Suppose that l also exists on the edge e' right of e . If both l' and l'' exist on e' , we would already have a counterexample on e' . Hence, at least one of the lines ends after e , that is, at least one of the good pairs results from inheritance after the edge e . On the other hand, this can only be the case for one of both, suppose for (l', l) . There has to be another good pair (l''', l) on e' , a contradiction to the choice of e . \square

Lemma 5. *There are only two possibilities to create a good pair (a, b) :*

- (i) *a and b start at the same node consecutively in the right order.*
- (ii) *A block crossing brings a and b together.*

Proof. During a and b exist, the good pair (a, b) can only be created by block crossings because either a and b have to cross each other or lines between a and b have to leave the path. Hence, (a, b) can only be created without a block crossing at the moment when the last of the two lines, say a , starts at a node v . In this case a has to be the first line starting at v on the top of P . This implies that by inheritance there is a good pair (c, b) , where c is the last line ending at v to the top. It follows that the good pair (c, b) , which is identical to (a, b) , existed before v . \square

In what follows, we will say that a solution (or algorithm) creates a good pair in a block crossing if the two lines of the good pair are brought together in the right order by that block crossing; analogously, we speak of breaking good pairs. It is easy to see that any solution, especially an optimal one, has to create all good pairs, and a block crossing can create at most three new pairs. There are only two possible ways for creating a good pair (a, b) : (i) a and b start at the same node consecutively in the right order, that is, they form an *initial good pair*, or (ii) a block crossing brings a and b together. Similarly, good pairs can only be destroyed by crossings before both lines end.

Using good pairs, we formulate our algorithm as follows; see Fig. 4(b) for an example.

We follow P from left to right. On an edge $e = (u, v)$ there are red lines that end at v to the top, green lines that end at v to the bottom, and black lines that continue on the next edge. We bring the red lines in the right order to the top by moving them upwards. Doing so, we keep existing good pairs together. If a line is to be moved, we consider the lines below it consecutively. As long as the current line forms a good pair with the next line, we extend the block that will be moved. We stop at the first line that does not

form a good pair with its successor. Finally we move the whole block of lines linked by good pairs in one block move to the top. Next, we bring the green lines in the right order to the bottom, again keeping existing good pairs together. There is an exception, where one good pair on e cannot be kept together. If the moved block is a sequence of lines containing both red and green lines, and possibly some—but not all—black lines, then it has to be broken, see Fig. 5. Note that this can only happen in the last move on an edge. There are two cases:

(i) A good pair in the sequence contains a black line and has been created by the algorithm previously. We break the sequence at this good pair.

(ii) All pairs with a black line are initial good pairs, that is, were not created by a crossing. We break at the pair that ends last of these. Inheritance is also considered, that is, a good pair ends only when the last of the pairs that are linked by inheritance ends.

After an edge has been processed, the lines ending to the top and to the bottom are on their respective side in the right relative order. Hence, our algorithm produces a feasible solution. We show that it produces a 3-approximation for the number of block crossings. A key property that we will show is that our strategy for case (ii) is optimal.

Theorem 3. *Let bc_{alg} and OPT be the number of block crossings created by the algorithm and an optimal solution, respectively. Then, $bc_{\text{alg}} \leq 3 \text{OPT}$.*

Proof. Normal block crossings, not breaking a good pair in the algorithm, always increase the number of good pairs. If we have a block crossing that breaks a good pair in a sequence as in case (i) then there has been a block crossing that created the good pair previously as a side effect, that is, there was an additional (red or green) good pair whose creation caused that block crossing. Hence, we can say that the destroyed good pair did not exist previously and still have at least one new good pair per block crossing.

If we are in case (ii), that is, all good pairs in the sequence are initial good pairs like in Fig. 5, then they also initially existed in the optimal solution. It is not possible to keep all those good pairs because the remaining black lines have to be somewhere between the block of red lines and the block of green lines. Hence, even the optimal solution has to break one of these good pairs on this edge or previously.

Let $\overline{bc}_{\text{alg}}$, $\overline{bc}_{\text{opt}}$ be the number of broken good pairs due to case (ii) in the algorithm and the optimal solution. In a crossing in which the algorithm breaks such a good pair the number of good pairs stays the same as one good pair is destroyed and another created. On the other hand, in a crossing that breaks a good pair, the number of good pairs can be increased by at most two even in the optimal solution (actually, it is not hard to see that it cannot be increased at all). Let gp be the total number of good pairs and let gp_{init} be the number of initial good pairs. Note that according to Def. 1 good pairs resulting from inheritance are not counted separately for gp as they are identified with another good pair. We get $gp \geq bc_{\text{alg}} - \overline{bc}_{\text{alg}} + gp_{\text{init}}$ and $gp \leq 3 \cdot \text{OPT} - \overline{bc}_{\text{opt}} + gp_{\text{init}}$. Hence, $bc_{\text{alg}} \leq 3 \text{OPT} + (\overline{bc}_{\text{alg}} - \overline{bc}_{\text{opt}})$ combining both estimates.

To prove the approximation factor 3 we only have to show that $\overline{bc}_{\text{alg}} \leq \overline{bc}_{\text{opt}}$. First, note that the edges where good pairs of case (ii) are destroyed, are exactly the edges where such a sequence of initial good pairs exists; that is, the edges are independent of any algorithm or solution. We show that, among these edges, our strategy ensures that the smallest number of pairs is destroyed, and pairs that are destroyed once are

reused as often as possible for breaking a sequence of initial good pairs. To this end, let $e'_1, \dots, e'_{\text{bc}_{\text{alg}}}$ be the sequence of edges, where the algorithm destroys a new good pair of type (ii), that is, an initial good pair that has never been destroyed before. We follow the sequence and argue that the optimal solution destroys a new pair for each of these edges. Otherwise, there is a pair e', e'' of edges in the sequence, where the optimal solution uses the same good pair p on both edges. Let p', p'' be the pairs used by the algorithm on e', e'' for breaking a sequence of initial good pairs. As p' was preferred by the algorithm over p , we know that p' still exists on e'' . As it is in a sequence with p , the algorithm does, therefore, still use p' on e'' , a contradiction completing the proof. \square

The algorithm needs $O(|L|(|L| + |E_P|))$ time. Note that it does normally not produce orderings with monotone block crossings. It can, however, be turned into a 3-approximation algorithm for MBCM. To this end, the definition of inheritance of good pairs, as well as the step of destroying good pairs has to be adjusted, and the analysis has to be improved.

Monotone Block Crossings on Paths We want to modify our algorithm so that it produces monotone block crossings which are a 3-approximation for the minimum number of monotone block crossings in $O(|L|(|L| + |E_P|))$ time. To this end, we first have to modify our definition of inheritance of good pairs such that we use only monotone block moves. More specifically, we prevent inheritance in some situations in which keeping a pair of lines together close to a vertex is not possible without having a forbidden crossing. We concentrate on inheritance with lines ending to the top; the other case is analogue.

Suppose we have a situation as shown in Fig. 6. Line c must not cross b . On the other hand it has to be below a_2 near node v and separate a_2 and b there. Hence, bringing or keeping a_2 and b together is of no value, as they have to be separated in any solution. We say that line b is *inheritance-preventing* at node v .

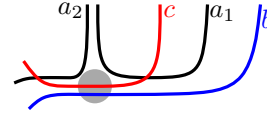


Fig. 6. Line c prevents that (a_2, b) inherits from (a_1, b) .

One part of our algorithm still needs to be changed in order to ensure monotonicity of the crossings. A block move including black lines could result in a forbidden crossing. We focus on the case, where black lines are moved together with red lines. This can only occur once per edge.

Let $r = b_0, b_1, \dots, b_k$ be the sequence of good pairs from the bottommost red line $r = b_0$ on. If there is some line l above the block that must not be crossed by a line b_i of the block, then we have to break the sequence. We consider such a case in which i is minimal. Hence, we have to break a good pair out of $(r, b_1), (b_1, b_2), \dots, (b_{i-1}, b_i)$. Similar to case (i) in the algorithm of the previous section, we break a pair of this sequence that is not initial. Otherwise (case (ii)), we choose the pair (b_{j-1}, b_j) with $j \leq i$ minimal such that the end node of b_j is below the path, and break the sequence there. Note that line l must end below the path, otherwise it would prevent inheritance of at least one of the good pairs in the sequence. Hence, also b_i ends below the path, and b_j is well-defined.

It is easy to see that our modified algorithm still produces a feasible ordering. We now show that it is also monotone.

Theorem 4. *The algorithm produces an ordering with monotone block crossings.*

Proof. We want to see that each block crossing is monotone, that is, a pair of lines that cross in a block crossing is in the wrong order before the crossing. Monotonicity of the whole solution then follows. We consider moves, where blocks of lines are brought to the top; the other case is analogue.

Suppose a red line r is brought to the top. As all red lines that have to leave above r are brought to the top before, r crosses only lines that leave below it, that is, lines that have to be crossed by r . If a black line l is brought to the top, then it is moved together in a block that contains a sequence of good pairs from the bottommost red line r' to l . Suppose l crosses a line c that should not be crossed by l . Line c cannot be red because all red lines that are not in the block that is moved at the moment have been brought to the top before. It follows that r' has to cross c . Hence, we can find a good pair (a, b) in the sequence from r' to l such that a has to cross c but b must not cross c . In this case, the algorithm will break at least one good pair between r' and b . It follows that c does not cross l , a contradiction. \square

Theorem 5. *Let bc_{alg} be the number of block crossings created by the algorithm and let OPT be the number of block crossings of an optimal solution using only monotone block moves. It holds that $\text{bc}_{\text{alg}} \leq 3 \text{OPT}$.*

Proof. As for non-monotone block crossings, all block crossings increase the number of good pairs, with the exception of breaking a sequence of initial good pairs in case (ii). Again, also the optimal solution has to have crossings, where such sequences are broken. In such a crossing, the two lines of the destroyed pair lose their partner. Hence, there is only one good pair after the crossing, and the number of good pairs does not change at all. Let $\overline{\text{bc}}_{\text{alg } t}$ be the number of splits for case (ii) where the block move brings lines to the top, and let $\overline{\text{bc}}_{\text{alg } b}$ be the number of such splits where the move brings lines to the bottom. We get

$$\begin{aligned} \text{bc}_{\text{alg}} &\leq 3 \cdot \text{OPT} + (\overline{\text{bc}}_{\text{alg}} - 3 \cdot \overline{\text{bc}}_{\text{opt}}) \\ &\leq 3 \cdot \text{OPT} + (\overline{\text{bc}}_{\text{alg } t} - \overline{\text{bc}}_{\text{opt}}) + (\overline{\text{bc}}_{\text{alg } b} - \overline{\text{bc}}_{\text{opt}}) \end{aligned}$$

To complete the proof, we show $\overline{\text{bc}}_{\text{alg } t} \leq \overline{\text{bc}}_{\text{opt}}$, and symmetrically $\overline{\text{bc}}_{\text{alg } b} \leq \overline{\text{bc}}_{\text{opt}}$.

Let $e'_1, \dots, e'_{\overline{\text{bc}}_{\text{alg } t}}$ be the sequence of edges, where the algorithm uses a new good pair, as a breakpoint for a sequence of type (ii) when lines leave to the top, that is, a good pair that has not been destroyed before. Again, we argue that even the optimal solution has to use a different breakpoint pair for each of these edges. Otherwise, there would be a pair e', e'' of edges in this sequence, where the optimal solution uses the same good pair p on both edges. Let p' and p'' be the two good pairs used by the algorithm on e' and e'' , respectively. Let $p' = (l', l'')$. We know that l' leaves the path to the top and l'' leaves to the bottom. Because all lines in the sequences on e' and e'' stay parallel, we know that lines above l' leave to the top, and lines below l'' leave to the bottom. Especially p' still exists on e'' , as p stays parallel and also still exists.

As in the description of the algorithm, let a and b be lines such that (a, b) is the topmost good pair in the sequence for which a line c exists on e'' that crosses a but not b . If (a, b) is below p' , then the algorithm would reuse p' instead of the new pair p'' , since (a, b) is in a sequence below p ; hence, also p' is in the sequence and above (a, b) .

Now suppose that (a, b) is above p' . The pair (a, b) is created by inheritance because c ends between a and b . As both a and b end to the top, separated from the bottom side of the path by p' , this inheritance takes place at a node, where a is the last line to end on the top side. But in this case c prevents the inheritance of the good pair (a, b) because it crosses only a , a contradiction. \square

4 Block Crossings on Trees

In what follows we focus on instances of (M)BCM that are trees. We first give an algorithm that bounds the number of block crossings. Then, we consider trees with an additional constraint on the lines; for these we develop a 6-approximation for MBCM.

Theorem 6. *For any tree T and lines L on T , we can order the lines with at most $2|L| - 3$ monotone block crossings in $O(|L|(|L| + |E|))$ time.*

Proof. We give an algorithm in which paths are inserted one by one into the current order; for each newly inserted path we create at most 2 monotone block crossings. The first line cannot create a crossing, and the second line crosses the first one at most once.

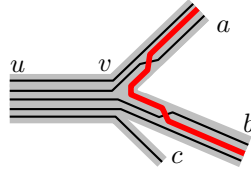


Fig. 7. Insertion of a new line (red) into the current order on edges va and vb .

We start at an edge incident to a terminal. Let L_{uv} be the lines passing through edge uv . When processing uv the paths L_{uv} are already in the correct order; they do not need to cross on yet unprocessed edges of T . We consider all unprocessed edges va, vb, \dots incident to v and build the correct order for them. The relative order of lines also passing through uv is kept unchanged. For all lines passing through v that were not treated before, we apply an insertion procedure, see Fig 7. Consider, e.g., the insertion of a line passing through va and vb . Close to v we add l on both edges at the innermost position such that we do not get vertex crossings with lines that pass through va or vb . We find its correct position in the current order of lines L_{va} close to a , and insert it using one block crossing. This crossing will be the last one on va going from v to a . Similarly, l is inserted into L_{vb} . We have to make sure that lines that do not have to cross are inserted in the right order. As we know the right relative order for a pair of such lines we can make sure that the one that has to be innermost at node v is inserted first.

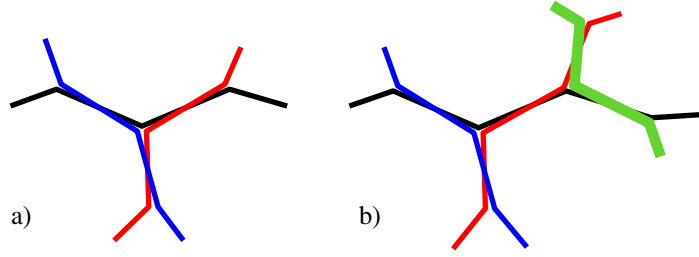


Fig. 8. Tree with $2|L| - 3$ necessary crossings for a) $|L| = 3$, b) $|L| = 4$.

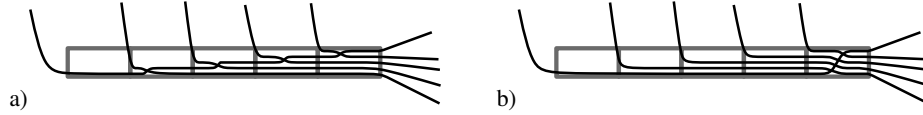


Fig. 9. a) The algorithm started leftmost produces 4 crossings; b) one block crossing suffices.

Similarly, by looking at the clockwise order of edges around v , we know the right order of line insertions such that there are no avoidable vertex crossings. When all new paths are inserted the orders on va, vb, \dots are correct; we proceed by recursively processing these edges.

When inserting a line, we create at most 2 block crossings, one per edge of l incident to v . After inserting the first two lines into the drawing there is at most one crossing. Hence, we get at most $2|L| - 3$ block crossings in total. Suppose monotonicity would be violated, that is, there is a pair of lines that crosses twice. The crossings then have been introduced when inserting the second of those lines on two edges incident to a node v . This can, however, not happen, as at node v the two edges are inserted in the right order. Hence, the block crossings of the solution are monotone. \square

In the following we show that the upper bound that our algorithm yields is tight.

Worst-Case Examples. Consider a graph show in Fig. 8. The new green path in Fig. 8b) is inserted so that it crosses 2 existing paths. This is the induction step for creating instances in which $2|L| - 3$ block crossings are necessary in any solution.

We also have a simple example in which the tree algorithm creates $|L| - 1$ crossings while a single block crossing suffices; see Fig. 9 for $|L| = 5$. The example can easily be extended to any number of lines. This shows that the algorithm does not yield a constant factor approximation.

The examples shows that the algorithm described in Theorem 6 does not guarantee an approximation of the optimal solution. Next, we introduce an additional constraint on the lines, which helps us to approximate the minimum number of block crossings.

Upward Trees We consider MBCM on an *upward tree* T , that is, a tree that has a planar upward drawing in which all paths are monotone in vertical direction, and all path sources are on the same height as well as all path sinks, see Fig. 10. Bekos et al. [3]

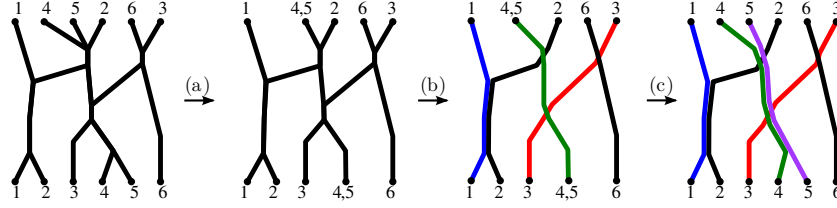


Fig. 10. Algorithm for upward trees: (a) simplification, (b) line ordering, (c) reinsertion.

already considered such trees (under the name “left-to-right trees”) for the metro line crossing minimization problem. Note that a graph whose skeleton is a path is not necessarily an upward tree. Our algorithm consists of three steps. First, we perform a simplification step removing some lines. Second, we use the algorithm for trees given in Section 4 on a simplified instance. Finally, we reinsert the removed lines into the constructed order. We first analyze the upward embedding.

Given an upward drawing of T , we read a permutation π produced by the terminals on the top; we assume that the terminals produce the identity permutation on the bottom. Similar to the single edge case the goal is to sort π by a shortest sequence of block moves. Edges of T restrict some block moves on π ; e.g., the blocks 1 4 and 5 in Fig. 10 cannot be exchanged as there is no suitable edge. However, we still can use the lower bound for block crossings on a single edge, see Section 2: For sorting a simple permutation π , at least $\text{bp}(\pi)/3$ block moves are necessary. We stress that simplicity of π is crucial here. To get an approximation, we show how to simplify a tree.

Consider two non-intersecting paths a and b that are adjacent in both permutations and share a common edge. We prove that one of these paths can be removed without changing the optimal number of block crossings. First, if any other line c crosses a then it also crosses b (i). This is implied by planarity and y -monotonicity of the drawing. Second, if c crosses both a and b then all three paths share a common edge (ii); otherwise, there would be a cycle due to planarity. Hence, for any solution for the paths $L - \{b\}$, we can construct a solution for L by inserting b without any new block crossings. To insert b , we must first move all block crossings on a to the common subpath with b . This is possible due to observation (ii). Finally, we can place b parallel to a .

To get a 6-approximation for an upward tree T , we first remove lines until the tree is simple. Then we apply the insertion algorithm presented in Section 4, and finally reinsert the lines removed in the first step. The number of block crossings is at most $2|L'|$, where L' is the set of lines of the simplified instance. As an optimal solution has at least $|L'|/3$ block crossings for this simple instance, and reinserting lines does not create new block crossings, we get

Theorem 7. *The algorithm yields a 6-approximation for MBCM on upward trees.*

5 Block Crossings on General Graphs

Finally, we consider general graphs. We suggest an algorithm that achieves an upper bound on the number of block crossings and show that it is asymptotically worst-case

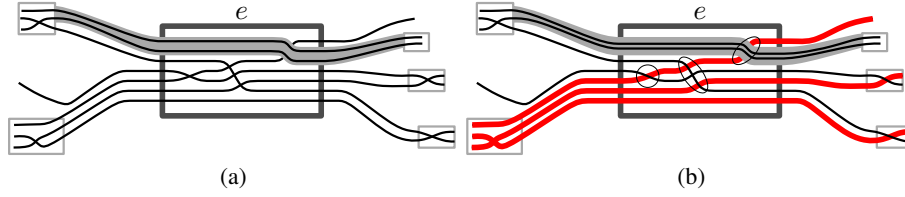


Fig. 11. Sorting the lines on edge e . (a) Cutting edges (marked) define groups. The lines marked in gray are merged as they are in the same group on both sides. (b) Sorting by insertion into the largest group (red); the merged lines always stay together, especially in their block crossing.

optimal. Our algorithm uses monotone block moves, that is, each pair of lines crosses at most once. The algorithm works on any embedded graph; it does not even need to be planar, we just need to know the circular order of incident edges around each vertex.

The idea of our algorithm is simple. We go through the edges in some arbitrary order. When we treat an edge, we completely sort the lines that traverse it. A crossing between a pair of lines can be created on the edge only if this edge is the first one treated by the algorithm that is used by both lines of the pair; see Algorithm 1. The crucial part is sorting the lines on an edge. Suppose we currently deal with edge e and want to sort L_e . Due to the path intersection property, the edge set used by the lines in L_e forms a tree on each side of e , see Fig. 11. We cut these trees at those edges that have already been processed by our algorithm. Now, each line on e starts at a leaf on one side and ends at a leaf on the other side. Note that multiple lines can start or end at the same leaf.

From the tree structure and the orderings on the edges processed previously, we get two orders of the lines, one on each side of e . We consider *groups of lines* that start or end at a common leaf of the tree (like the red lines in Fig. 11). All lines of a group have been seen on a common edge, and, hence, have been sorted. Therefore lines of the same group form a consecutive subsequence on one side of e , and have the same relative order on the other side of e .

Let g and g' be a group of lines on the left and on the right side of e , respectively. Suppose the set L' of lines starting in g and ending in g' consists of multiple lines. As the lines of g as well as the lines of g' stay parallel on e , L' must form a consecutive subsequence (in the same order) on both sides. Now we *merge* L' into one representative for the sequence of lines, that is, we remove all lines of L' and replace them by a single line that is in the position of the lines of L' on the sequences on both sides of e . Once we find a solution, we replace the representative by the sequence without changing the

```

foreach edge  $e$  with  $|L_e| > 1$  do
    Build order of lines on both sides of  $e$ 
    Merge lines that are in the same group on both sides
    Find the largest group of consecutive lines that stay parallel on  $e$ 
    Insert all other lines into this group and undo merging

```

Algorithm 1: Ordering the lines on a graph

number of block crossings. Consider a crossing that involves the representative of L' , that is, it is part of one of the moved blocks. After replacing it, the sequence L' of parallel lines is completely contained in the same block. Hence, we do not need additional block crossings.

We apply this merging for all pairs of groups on the left and right end of E . Then, we identify a group with the largest number of lines after merging, and insert all remaining lines into it one by one. Clearly, each insertion requires at most one block crossing; in Fig. 11 we need three block crossings to insert the lines into the largest (red) group. After computing the crossings, we undo the merging step and get a solution for edge e .

Theorem 8. *Algorithm 1 sorts all lines in $O(|E|^2|L|)$ time by monotone block moves. The resulting number of block crossings is $O(|L|\sqrt{|E'|})$, where E' is the set of edges with at least two lines on them.*

Proof. First, it is easy to see that no unnecessary crossings are created. Additionally, we care about all edges with at least two lines, which ensures that all necessary crossings will be placed. Hence, we get a feasible solution using monotone crossings. Our algorithm sorts the lines on an edge in $O(|L||E|)$ time. We can build the tree structure and find the orders and groups by following all lines until we find a terminal or an edge that was processed before in $O(|L||E|)$ time. Merging lines and finding the largest group need $O(|L|)$ time; sorting by insertion into this group and undoing the merging can be done in $O(|L|^2)$ time. Note that $|L| \leq |E|$ due to the path terminal property.

For analyzing the total number of block crossings, we maintain an *information* table T with $|L|^2$ entries. Initially, all the entries are empty. After processing an edge e in our algorithm, we fill entries $T[l, l']$ of the table for each pair (l, l') of lines that we see together for the first time. The main idea is that with b_e block crossings on edge e we fill at least b_e^2 new entries of T . The upper bound then can be concluded.

More precisely, let the *information gain* $I(e)$ be the number of pairs of (not necessarily distinct) lines l, l' that we see together on a common edge e for the first time. Clearly, $\sum_{e \in E} I(e) \leq |L|^2$. Suppose that $b_e^2 \leq I(e)$ for each edge e . Then, $\sum_{e \in E} b_e^2 \leq \sum_{e \in E} I(e) \leq |L|^2$. Using the Cauchy-Schwarz inequality $|\langle x, y \rangle| \leq \sqrt{\langle x, x \rangle \cdot \langle y, y \rangle}$ with x as the vector of the b_e and y as a vector of 1-entries, we see that the total number of block crossings is $\sum_{e \in E'} b_e \leq |L|\sqrt{|E'|}$.

We still have to show that $b_e^2 \leq I(e)$ for an edge e . For doing so, we analyze the lines after the merging step. Consider the groups on both sides of e ; we number the groups on the left side $\mathfrak{L}_1, \dots, \mathfrak{L}_n$ and the groups on the right side $\mathfrak{R}_1, \dots, \mathfrak{R}_m$ with $l_i = |\mathfrak{L}_i|, r_j = |\mathfrak{R}_j|$ for $1 \leq i \leq n, 1 \leq j \leq m$. Without loss of generality, we can assume that \mathfrak{L}_1 is the largest group into which all remaining lines are inserted. Then, $b_e \leq |L_e| - l_1$. Let s_{ij} be the number of lines that are in group \mathfrak{L}_i on the left side and in group \mathfrak{R}_j on the right side of e . Note that $s_{ij} \in \{0, 1\}$, otherwise we could still merge lines. Then $l_i = \sum_j s_{ij}, r_j = \sum_i s_{ij}, s := |L_e| = \sum_{ij} s_{ij}$, and $b_e = s - l_1$. The information gain is $I(e) = s^2 - \sum_i l_i^2 - \sum_j r_j^2 + \sum_{ij} s_{ij}^2$. By applying Lemma 6 we get $b_e^2 \leq I(e)$. To complete the proof, note that the unmerging step cannot decrease $I(e)$. \square

Lemma 6. *For $1 \leq i \leq n$ and $1 \leq j \leq m$ let $s_{ij} \in \{0, 1\}$. Let $l_i = \sum_j s_{ij}$ for $1 \leq i \leq n$ and let $r_j = \sum_i s_{ij}$ for $1 \leq j \leq m$ such that $l_1 \geq l_i$ for $1 \leq i \leq n$ and $l_1 \geq r_j$ for*

$1 \leq j \leq m$. Let $s = \sum_{i=1}^n \sum_{j=1}^m s_{ij}$, $b = s - l_1$, and $I = s^2 - \sum_i l_i^2 - \sum_j r_j^2 + \sum_{ij} s_{ij}^2$. Then $b^2 \leq I$.

Proof. It is easy to check that for any i, j it holds that $s_{ij}(s_{ij} - s_{1j}) \geq 0$ as $s_{ij} \in \{0, 1\}$. Using this property, we see that

$$\begin{aligned}
I - b^2 &= s^2 - \sum_{i=1}^n l_i^2 - \sum_{j=1}^m r_j^2 + \sum_{i=1}^n \sum_{j=1}^m s_{ij}^2 - s^2 + 2sl_1 - l_1^2 \\
&= \sum_{i=1}^n \sum_{j=1}^m s_{ij}^2 + 2l_1(s - l_1) - \sum_{i=2}^n l_i^2 - \sum_{j=1}^m r_j \sum_{i=1}^n s_{ij} \\
&= \sum_{i=1}^n \sum_{j=1}^m s_{ij}^2 + 2l_1 \sum_{i=2}^n \sum_{j=1}^m s_{ij} - \sum_{i=2}^n l_i \sum_{j=1}^m s_{ij} - \sum_{i=1}^n \sum_{j=1}^m s_{ij} r_j \\
&= \sum_{i=2}^n \sum_{j=1}^m s_{ij} (s_{ij} + 2l_1 - l_i - r_j) - \sum_{j=1}^m s_{1j} \sum_{i=2}^n s_{ij} \\
&\geq \sum_{i=2}^n \sum_{j=1}^m s_{ij} (s_{ij} - s_{1j}) \geq 0.
\end{aligned}$$

□

We can show that the upper bound on the number of block crossings that our algorithm achieves is tight. To this end, we use the existence of special Steiner systems for building (non-planar) worst-case examples of arbitrary size in which many block crossings are necessary; see Theorem 9.

Theorem 9. *There exists an infinite family of graphs $G = (V, E)$ with set of lines L so that $\Omega(|L|\sqrt{|E'|})$ block crossings are necessary in any solution, where E' is the set of edges with at least two lines on them.*

Proof. From the area of projective planes it is known that, for any prime power q , a $S(q^2 + q + 1, q + 1, 2)$ Steiner system exists [15], that is, there is a set \mathcal{S} of $q^2 + q + 1$ elements with subsets $S_1, S_2, \dots, S_{q^2+q+1}$ of size $q + 1$ such that any pair of elements $s, s' \in \mathcal{S}$ appears together in exactly one set S_i .

We build a graph $G = (V, E)$ by first adding vertices s_1, s_2 and an edge (s_1, s_2) for any $s \in \mathcal{S}$. These edges will be the only ones with multiple lines on them, that is, E' . Additionally, we add an edge s_2, s'_1 for any pair $s, s' \in \mathcal{S}$. Next, we build a line l_i for any set S_i as follows. We take some arbitrary order $s, s', s'', \dots, s^{(q)}$ on the elements of S_i , and build the path $s(l_i), s_1, s_2, s'_1, s'_2, s''_1, \dots, s^{(q)}_2, t(l_i)$ with extra terminals $s(l_i)$ and $t(l_i)$ in which l_i starts and ends, respectively; see Fig. 12(a). As any pair of lines shares exactly one edge the path intersection property holds. We order the edges around vertices s_1 and s_2 so that all $q + 1$ lines on the edge representing any $s \in \mathcal{S}$ have to cross by making sure that the orders of lines in s_1 and s_2 are exactly reversed; see Fig. 12(b). Then, q block crossings are necessary on each edge, and, hence, $(q^2 + q + 1)q = \theta(q^3)$ block crossings in total. On the other hand, $|L|\sqrt{|E'|} = (q^2 + q + 1)\sqrt{q^2 + q + 1} = \theta(q^3)$. Note that the graph G is not planar. □

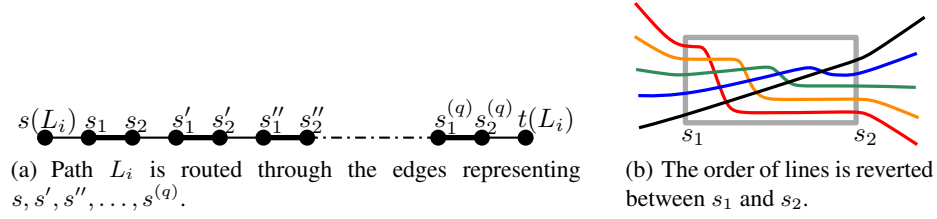


Fig. 12. Construction of the worst-case example.

6 Future Work

Our algorithm for general graphs cannot be applied if lines are more complex subgraphs than paths, or if the path intersection property does not hold. On the other hand, in many metro networks, there are just few lines violating these properties. We suggest to first create an instance with our properties by deleting few (parts of) lines. Then, after applying our algorithm, the deleted parts can be reinserted by keeping them parallel to other lines and reusing crossings as often as possible.

Another practical problem is the distribution of block crossings. In our opinion, crossings of lines should preferably be close to the end of their common subpath as this makes it easier to recognize that the lines do cross. For making a metro line easy to follow the important criterion is the number of its bends. Hence, an interesting question is how to sort metro lines using the minimum total number of bends.

From a theoretical point of view, the complexity status of MBCM on a single edge is an interesting open problem.

Acknowledgements

We are grateful to Sergey Bereg, Alexander E. Holroyd, and Lev Nachmanson for the initial discussion of the block crossing minimization problem, and for pointing out a connection with sorting by transpositions. We thank Jan-Henrik Haunert, Joachim Spoerhase, and Alexander Wolff for fruitful discussions and suggestions.

References

1. E. N. Argyriou, M. A. Bekos, M. Kaufmann, and A. Symvonis. On metro-line crossing minimization. *J. Graph Algorithms Appl.*, 14(1):75–96, 2010.
2. V. Bafna and P. A. Pevzner. Sorting by transpositions. *J. Discr. Math.*, 11:224–240, 1998.
3. M. A. Bekos, M. Kaufmann, K. Potika, and A. Symvonis. Line crossing minimization on metro maps. In *Graph Drawing*, pages 231–242, 2008.
4. M. Benkert, M. Nöllenburg, T. Uno, and A. Wolff. Minimizing intra-edge crossings in wiring diagrams and public transport maps. In *Graph Drawing*, pages 270–281, 2007.
5. L. Bulteau, G. Fertin, and I. Rusu. Sorting by transpositions is difficult. *SIAM J. Discr. Math.*, 26(3):1148–1180, 2012.

6. D. A. Christie and R. W. Irving. Sorting strings by reversals and by transpositions. *SIAM J. Discr. Math.*, 14(2):193–206, 2001.
7. I. Elias and T. Hartman. A 1.375-approximation algorithm for sorting by transpositions. *IEEE/ACM Trans. Comput. Biol. Bioinformatics*, 3(4):369–379, 2006.
8. G. Fertin, A. Labarre, I. Rusu, E. Tannier, and S. Vialette. *Combinatorics of Genome Rearrangements*. The MIT Press, 2009.
9. P. Groeneveld. Wire ordering for detailed routing. *IEEE Des. Test*, 6:6–17, 1989.
10. L. S. Heath and J. P. C. Vergara. Sorting by bounded block-moves. *Discrete Applied Mathematics*, 88(1–3):181–206, 1998.
11. M. Marek-Sadowska and M. Sarrafzadeh. The crossing distribution problem. *IEEE Trans. CAD Integrated Circuits Syst.*, 14(4):423–433, 1995.
12. M. Nöllenburg. An improved algorithm for the metro-line crossing minimization problem. In *Graph Drawing*, pages 381–392, 2009.
13. S. Pupyrev, L. Nachmanson, S. Bereg, and A. E. Holroyd. Edge routing with ordered bundles. In *Graph Drawing*, pages 136–147, 2011.
14. F. Schreiber. High quality visualization of biochemical pathways in BioPath. *In Silico Biology*, 2(2):59–73, 2002.
15. O. Veblen and W. H. Bussey. Finite projective geometries. *Trans. AMS*, 7(2):241–259, 1906.